Processes

CS 105: Computer Systems
Lecture 9

Melissa O'Neill

February 18, 2026

Learning Goals

- **Describe the two major abstractions a process provides**
- **Explain what a process context is and the purpose of *context switching***
- **Build a process graph for a program involving `forks` and identify feasible and infeasible output**

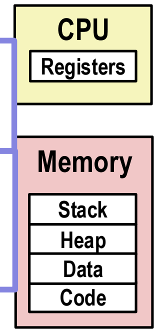
Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Important: a process is *not the same as a program*

Demo – process = program in execution

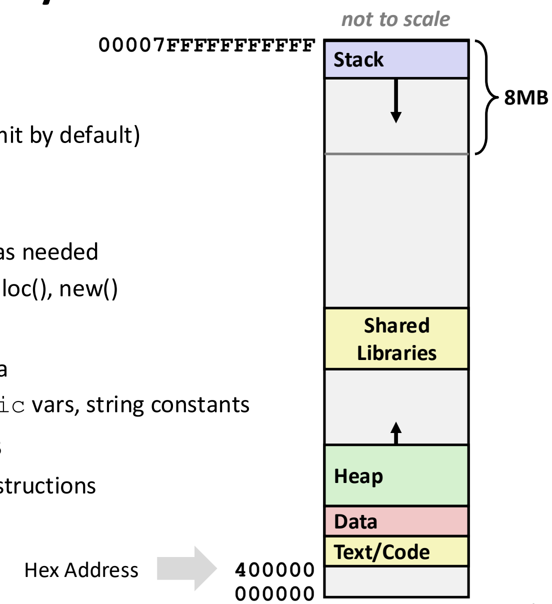
Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Important: a process is *not the same as a program*
- **Process provides each program with two key abstractions:**
  - ***Logical control flow*** **today**
    - Each program seems to have exclusive use of the CPU
    - Provided by mechanism called *context switching*
  - ***Private address space*** **future lecture**
    - Each program seems to have exclusive use of main memory.
    - Provided by mechanism called *virtual memory*



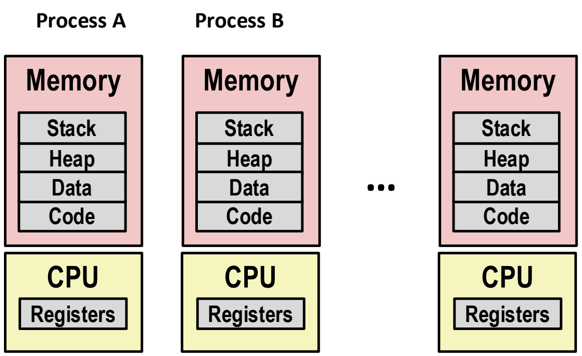
x86-64 Memory Layout

- **Stack**
  - Runtime stack (8MB limit by default)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text , Shared Libraries**
  - Executable machine instructions
  - Read-only

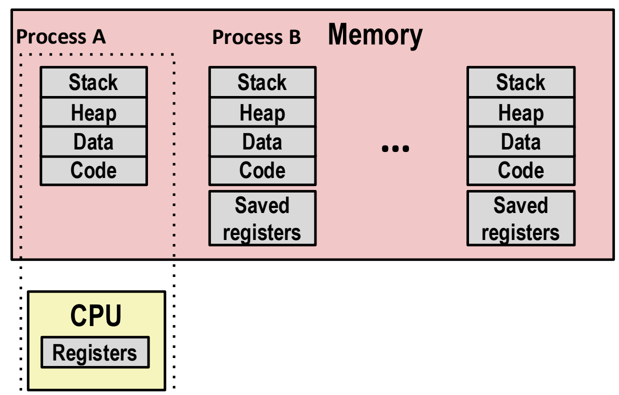


Multiprocessing: The Illusion

each process has its own CPU/memory

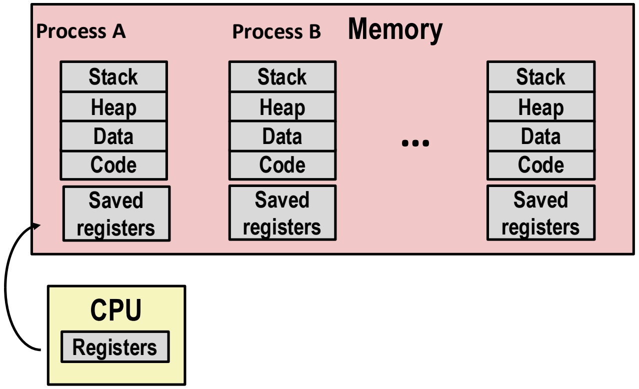


Multiprocessing: The (Single core) Reality



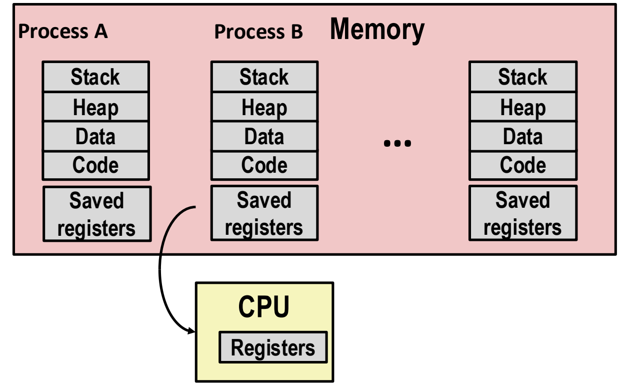
- **Process A runs for awhile**

Multiprocessing: The (Single core) Reality



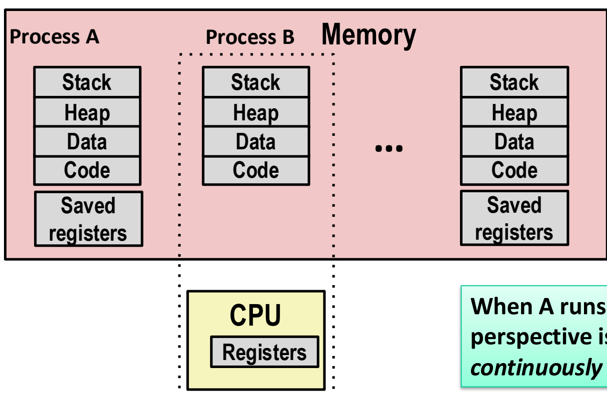
- **Process A runs for awhile** → **Context switch:**
  A stopped, A's registers saved in memory

Multiprocessing: The (Single core) Reality



- **Process A runs for awhile** → **Context switch:**
  A stopped, A's registers saved in memory
  B's registers restored

Multiprocessing: The (Single core) Reality



When A runs again… its perspective is that *it runs continuously* and owns the CPU

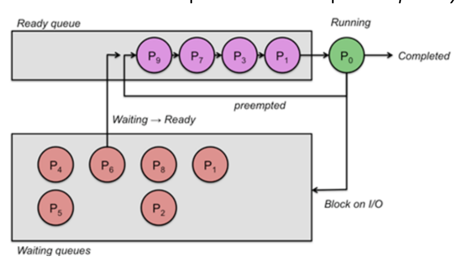
- **Process A runs for awhile** → **Context switch:**
- **Process B runs for awhile** ← **A stopped, A's registers saved in memory**
  B's registers restored

An Aside: Scheduling Processes

Operating system *process scheduler* decides when processes can run
- A preemptive scheduler can suspend a running process to allow another process to run

Important considerations
- Fairness, no starvation
- Efficiency: keep CPUs busy
- Process response time and throughput

Example: round-robin scheduling policy
- Preemptive version of first in, first out (FIFO)
- Process preempted after it exceeds some amount of time, called a time slice or quantum
- Can add different queues to handle process *priority*

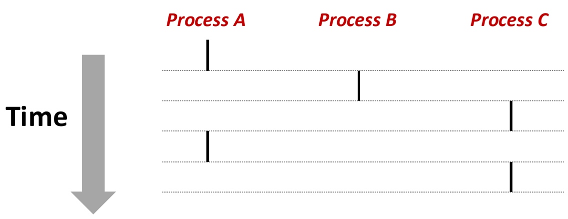


Concurrent Processes

- Each process is a logical control flow.
- Two processes *run **concurrently*** (*are concurrent)* if their flows overlap in time
- Otherwise, they are ***sequential***

- Examples (running on single core):

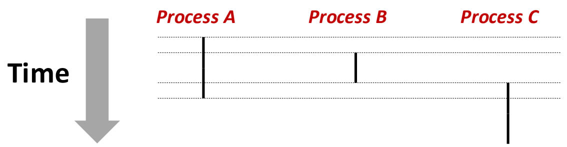
Which pairs of processes are concurrent?
Which are sequential?



User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other

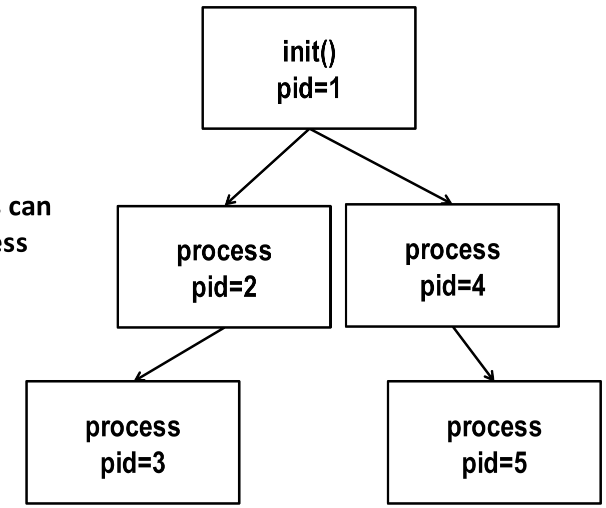


Creating a process

Every process has a unique process id (pid)

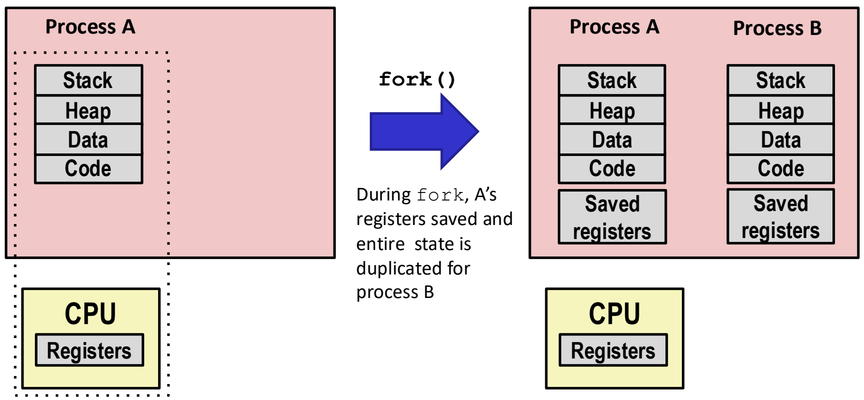
- System startup

- Then any process can start a *new* process
  - fork
  - fork → execve



Creating a process: `fork`

Process A running just before `fork`



During `fork`, A's registers saved and entire state is duplicated for process B

After `fork`, either Process A or B can be chosen to run
…or kernel can choose to context switch to a different process

Example: fork

Parent: pVal = m

```
int main()
{
   pid_t pVal;
   int x = 1;

   pVal = fork();
   if (pVal == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

After fork, parent and child have distinct memories; i.e., each has its own copy of *pVal* and *x*.

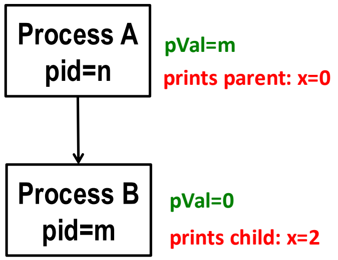
Child: pVal = 0

```
int main()
{
   pid_t pVal;
   int x = 1;

   pVal = fork();
   (pVal == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

fork called once, returns twice!
Return values:
- In parent: fork returns pid of child
- In child: fork returns 0

Example: fork

```
int main()
{
   pid_t pVal;
   int x = 1;

   pVal = fork();
   if (pVal == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
                                    fork.c
```



What will have printed to the console after both A and B finish?

```
linux> ./fork
parent: x=0
child : x=2
```

or

```
linux> ./fork
child : x=2
parent: x=0
```


Example: fork *Process Graph*

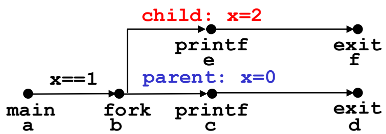
- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each **vertex** is the execution of a statement; first vertex can be function name
  - Each graph begins with a vertex with no in-edges
  - printf vertices can be labeled with output
  - A **directed edge** from a to b means *a happens before b*
  - Edges can be labeled with current value of variables

```
int main()
{
  pid_t pVal;
  int x = 1;

  pVal = fork();
  if (pVal == 0) {  /* Child */
    printf("child : x=%d\n", ++x);
    exit(0);
   }

  /* Parent */
  printf("parent: x=%d\n", --x);
  exit(0);
}
                                    fork.c
```

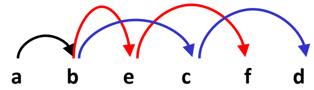

Example: fork *Process Graph*

Interpreting Process Graphs

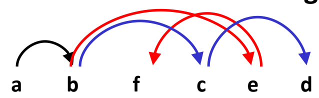
- Any *topological sort* of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right



Feasible total ordering:



Infeasible total ordering:



Exercise

First draw the process graph for this function.
Then give one feasible output and one infeasible output (i.e., what would be printed to the screen).

```
void fork2()
{
   printf("L0\n");
   fork();
   printf("L1\n");
   fork();
   printf("Bye\n");
}                    forks.c
```


`fork` summary

```
int main()
{
   pid_t pVal;
   int x = 1;

   pVal = fork();
   if (pVal == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}                    fork.c
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files (more later)
  - `stdout` is the same in both parent and child

Terminated processes

A terminated process exists on the system until it is killed.

- When a child process terminates:

i. Child terminates

Child → Parent

ii. Kernel reports exit status of child to parent

iii. Parent receives report

iv. Kernel kills child process

- Parent needs to ask for and receive exit status of children
  - Called "reaping"
  - Terminated child that hasn't been reaped is called a zombie process
- If parent terminates without reaping its children, the `init` process can reap the children
  - Long-running parent processes that don't reap yield lots of zombies!

Zombie Example

```
void fork7() {
  if (fork() == 0) {
    /* Child */
    printf("Terminating Child, PID = %d\n", getpid());
    exit(0);
  } else {
    printf("Running Parent, PID = %d\n", getpid());
    while (1); /* Infinite loop */
  }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by `init`

`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the `pid` of the child process that terminated
- (see textbook for more variants of `wait`)

`wait`: Synchronizing with Children

```
void fork9() {
 int child_status;

 if (fork() == 0) {
  printf("HC: hello from child\n")
  exit(0);

 } else {
   printf("HP: hello from parent\n");
   wait(&child_status);
   printf("CT: child has terminated\n");
 }
 printf("Bye\n");
}
```

forks.c